RIMS Kôkyûroku Bessatsu
B54 (2015), 71–98

Interval Arithmetic and Its Implementations

By

Naoya YAMANAKA* and Shin'ichi OISHI**

Abstract

A calculation method that derives mathematically correct results through numerical calculation, considering all the possible errors such as rounding errors and truncation errors, is called *"verified numerical computation"*. Recently, as a wide range of research activities on the verified numerical computation for various problems have been promoted by research groups in Japan such as those in Kyushu University and Waseda University, the calculation method has been widely acknowledged. This article describes specific implementation methods and notes on them, regarding the calculation methods in the interval operations, which are both the basis and basic operations of verified numerical computation.

§ 1. Intervals and Real Interval Operations

As any numerical calculation cannot directly handle real numbers, many calculations involve an error, which has been acknowledged as a problem since the beginning of the birth of electronic computers. As a result, considerable expertise has been garnered regarding the stability or other properties of the calculation methods, and many excellent ones have been designed to have the least possible influence from errors. However, it is difficult to fully estimate the impacts from any round error in actual calculations, and most of the existing calculation methods cannot provide any reliable validation to the accuracy of calculation results.

As a method that can bring reliable validation regarding the impacts of errors, an interval approach has been known. As any computer that uses floating-point numbers cannot represent all of the real numbers, it is inevitable that input numerical values and calculations are rounded. For this reason, the concept of interval is naturally introduced

Received December 31, 2014. Accepted February 2, 2015.
2010 Mathematics Subject Classification(s):
Key Words: *interval arithmetic, verified numerical computation.*
*Teikyo Heisei University, Tokyo, 164-8530, Japan / CREST, JST, Saitama, 332-0012, Japan.
e-mail: n.yamanaka@thu.ac.jp
**Waseda University, Tokyo 169-8555, Japan / CREST, JST, Saitama, 332-0012, Japan.
e-mail: oishi@waseda.jp

to hold the correct mathematical calculations in computers. An interval represents a set of numbers and at the same time all the numbers in the interval.

The contents of this article are written referring to some papers and books about *verified numerical computations*: for example, [1], [2] and [3].

Definition 1. *For any $x \in \mathbb{R}$, an interval is the closed interval represented with the following:*

$$[\underline{x},\ \overline{x}] = \{x \mid \underline{x} \leqq x \leqq \overline{x}\}, \tag{1.1}$$

where, $\underline{x} \leqq \overline{x} \in \mathbb{R}$.

The symbols $\underline{x}$ and $\overline{x}$ are referred to as the lower end and upper end of the interval, respectively. For simplicity, $[x]$ shall hereinafter represent the interval that satisfies the following:

$$[x] = [\underline{x},\ \overline{x}]\,. \tag{1.2}$$

Let $\mathbb{IR}$ be a set of intervals whose elements are real numbers and satisfy the following:

$$\mathbb{IR} = \{[x] \,|\underline{x},\ \overline{x} \in \mathbb{R}\}. \tag{1.3}$$

The operations on such intervals shall be defined as follows:

Definition 2. *Given any two intervals $[x], [y] \in \mathbb{IR}$, the binary operation on them shall be defined as follows:*

$$[x] \circ [y] = \{x \circ y \mid \textit{for } \forall x \in [x],\ \forall y \in [y]\} \tag{1.4}$$

Similarly, the unary operation $f(x)$ shall be defined as follows:

$$f([x]) = \{f(x) \mid \textit{for } \forall x \in [x]\} \tag{1.5}$$

These operations are referred to as real interval operations.

The four arithmetic operations on the intervals can be represented as shown in **Algorithm 1** according to **Definition 2**. **Algorithm 1** indicates that any interval operation can be executed with a finite number of real number operations. In addition,

Algorithm 1 Four arithmetic operations between intervals

Calculation methods for four arithmetic operations on any intervals $[x],\ [y] \in \mathbb{IR}$ for any binary operation $\circ \in \{+, -, \times, /\}$:

$$[x] \circ [y] = [\min(\underline{x} \circ \underline{y}, \underline{x} \circ \overline{y}, \overline{x} \circ \underline{y}, \overline{x} \circ \overline{y}),\ \max(\underline{x} \circ \underline{y}, \underline{x} \circ \overline{y}, \overline{x} \circ \underline{y}, \overline{x} \circ \overline{y})]. \tag{1.6}$$

$$[x] + [y] = [\underline{x} + \underline{y},\ \overline{x} + \overline{y}] \tag{1.7}$$

$$[x] - [y] = [\underline{x} - \overline{y},\ \overline{x} - \underline{y}] \tag{1.8}$$

$$[x] \times [y] = [\min\left(\underline{x}\underline{y}, \overline{x}\overline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}\right),\ \left(\underline{x}\underline{y}, \overline{x}\overline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}\right)] \tag{1.9}$$

$$[x] \,/\, [y] = [\min\left(\underline{x}/\underline{y}, \overline{x}/\overline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}\right),\ \max\left(\underline{x}/\underline{y}, \overline{x}/\overline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}\right)] \qquad (0 \notin [y]) \tag{1.10}$$

Algorithm 2 Real interval multiplication method for any intervals $[x], [y] \in \mathbb{IR}$

	$[x] < 0$	$[x] \ni 0$	$[x] > 0$
$[y] < 0$	$[\overline{x}\overline{y},\ \underline{x}\underline{y}]$	$[\overline{x}\underline{y},\ \underline{x}\underline{y}]$	$[\overline{x}\underline{y},\ \underline{x}\overline{y}]$
$[y] \ni 0$	$[\underline{x}\overline{y},\ \underline{x}\underline{y}]$	$[\min(\underline{x}\overline{y},\ \overline{x}\underline{y}),\ \max(\underline{x}\underline{y},\ \overline{x}\overline{y})]$	$[\overline{x}\underline{y},\ \overline{x}\overline{y}]$
$[y] > 0$	$[\underline{x}\overline{y},\ \overline{x}\underline{y}]$	$[\underline{x}\overline{y},\ \overline{x}\overline{y}]$	$[\underline{x}\underline{y},\ \overline{x}\overline{y}]$

Algorithm 3 Real interval division method for any intervals $[x], [y] \in \mathbb{IR}$

	$[x] < 0$	$[x] \ni 0$	$[x] > 0$
$[y] < 0$	$[\overline{x}/\underline{y},\ \underline{x}/\overline{y}]$	$[\overline{x}/\overline{y},\ \underline{x}/\overline{y}]$	$[\overline{x}/\overline{y},\ \underline{x}/\underline{y}]$
$[y] > 0$	$[\underline{x}/\underline{y},\ \overline{x}/\overline{y}]$	$[\underline{x}/\underline{y},\ \overline{x}/\underline{y}]$	$[\underline{x}/\overline{y},\ \overline{x}/\underline{y}]$

the case calculations can be used in multiplication and division. **Algorithm 2** and **Algorithm 3** show such calculations.

The interval arithmetic satisfies the monotonicity in inclusion relationship:

$$[x] \subseteq [x'], [y] \subseteq [y'] \Longrightarrow [x] \circ [y] \subseteq [x'] \circ [y'], \qquad \circ \in \{+, -, \times, /\}. \tag{1.11}$$

The commutative law and the associative law hold true for additions and multiplications:

$$[x] \circ [y] = [y] \circ [x], \qquad \circ \in \{+, \times\} \tag{1.12}$$

$$[x] \circ ([y] \circ [z]) = ([x] \circ [y]) \circ [z]. \qquad \circ \in \{+, \times\} \tag{1.13}$$

However, the inverse elements in addition and multiplication do not exist. Indeed, $-[x] = [-\overline{x},\ -\underline{x}]$ holds true, and the following also hold true:

$$0 = [0] \subseteq [x] - [x] = [\underline{x} - \overline{x},\ \overline{x} - \underline{x}], \tag{1.14}$$

$$1 = [1] \subseteq [x]/[x]. \tag{1.15}$$

The equal signs in the inclusions above hold only if $[x]$ is a point interval.

In addition, the distributive law is not true of the interval operations. Instead, the following sub-distributive law holds:

$$[x] \times ([y] + [z]) \subseteq [x] \times [y] + [x] \times [z]. \tag{1.16}$$

The equal sign in this expression holds true, for example, when the intervals $[y]$ and $[z]$ have the same sign.

This article calls an interval representation using the upper and lower ends of the interval an inf-sup type interval. In addition to the inf-sup type, the interval representation includes the mid-rad type interval using a center and radius. A mid-rad type interval is the closed interval that is represented for any $x \in \mathbb{R}$ as follows:

The following relationship exists between mid-rad type intervals and inf-sup type intervals:

$$[x] = [\underline{x},\ \overline{x}] = [x_c - x_r,\ x_c + x_r] = \left\langle \frac{\overline{x} + \underline{x}}{2}, \frac{\overline{x} - \underline{x}}{2} \right\rangle = \langle x_c,\ x_r \rangle = \langle x \rangle\,. \tag{1.17}$$

§ 2. Interval Arithmetic for inf-sup Type with Double-Precision Floating-Point Numbers

This article has been prepared on the basis of floating-point numbers in compliance with IEEE Standard 754, which are used in most numerical calculations with current computers. The IEEE Standard 754 is one of the standards for floating-point numbers and defines items such as "representation of numbers in computers", "precisions of calculations of four arithmetic operations and square root", "rounding" and "exceptions". In order to implement intervals using floating-point numbers on computers, a method that uses a changed rounding mode has been used widely. The rounding is one of the parameters for floating-point arithmetic specified by IEEE Standard 754, and specifies which floating-point number must be returned for a mathematical calculation. IEEE

Standard 754 defines four rounding modes: round to nearest, round upward, round downward and round toward 0; and it is widely known that a smallest interval including the mathematical result is found if any one of the four arithmetic operations is executed through the round upward and round downward mode (Section 2.1). But, some computational environments only accept the round-to-nearest mode. Even if a environment is changeable, a changing takes a considerable execution time compared with the execution time of one floating-point operation. In such cases, interval operation methods using only the round-to-nearest are available. The interval operation methods using only round-to-nearest are roughly divided into two types of calculation method: one returns the same interval width as interval operation methods using round upward or round downward (Section 2.2) and the other is capable of rapid calculation that returns a larger interval width than methods using either of the rounding modes changed from round-to-nearest (Section 2.3).

Let $\mathbb{F}$ be a set of double-precision floating-point numbers conforming to IEEE Standard 754. Also, let $\mathbb{IF}$ be a set of inf-sup intervals that have floating-point numbers as their elements:

$$\mathbb{IF} = \{[x] \in \mathbb{IR} \,|\underline{x},\ \overline{x} \in \mathbb{F}\}. \tag{2.1}$$

Given any two intervals $[x], [y] \in \mathbb{IF}$, the binary operation on them shall be defined as follows:

$$[x] \circ [y] \supseteq \{x \circ y \mid \text{for } \forall x \in [x],\ \forall y \in [y]\}. \tag{2.2}$$

Similarly, the unary operation $f(x)$ shall be defined as follows:

$$f([x]) \supseteq \{f(x) \mid \text{for } \forall x \in [x]\}. \tag{2.3}$$

These operations are referred to as interval operations.

Note that the basic knowledge of the floating-point numbers and their operations are summarized in Appendix A.

§ 2.1. Implementation 1 : Interval Arithmetic of inf-sup Type using Changed Rounding

Using round upward or round downward enables four arithmetic operations to give a result that satisfies the expression (2.2). **Algorithm 4**–**Algorithm 7** show four arithmetic operations using round upward and round downward. Note that the following holds true for any binary operation $\circ \in \{+, -, \times, \div\}$ and any inf-sup type intervals $[x]$ and $[y]$ in **Algorithm 4**–**Algorithm 7**:

$$[z] \supseteq [x] \circ [y]. \tag{2.4}$$

Algorithm 4 Interval addition using changed rounding modes

function $z = \mathrm{Addition}(x, y)$
$\underline{z} = \underline{x} \check{+} \underline{y};$
$\overline{z} = \overline{x} \hat{+} \overline{y};$
end

Algorithm 5 Interval subtraction using changed rounding modes

function $z = \mathrm{Subtraction}(x, y)$
$\underline{z} = \underline{x} \check{-} \overline{y};$
$\overline{z} = \overline{x} \hat{-} \underline{y};$
end

Algorithm 6 Interval multiplication using changed rounding modes

function $z = \mathrm{Multiplication}(x, y)$
$\underline{z} = \min\left(\underline{x} \check{\times} \underline{y}, \overline{x} \check{\times} \overline{y}, \underline{x} \check{\times} \overline{y}, \overline{x} \check{\times} \underline{y}\right);$
$\overline{z} = \max\left(\underline{x} \hat{\times} \underline{y}, \overline{x} \hat{\times} \overline{y}, \underline{x} \hat{\times} \overline{y}, \overline{x} \hat{\times} \underline{y}\right);$
end

Algorithm 7 Interval division using changed rounding modes

function $z = \mathrm{Division}(x, y)$
$\underline{z} = \min\left(\underline{x} \check{/} \underline{y}, \overline{x} \check{/} \overline{y}, \underline{x} \check{/} \overline{y}, \overline{x} \check{/} \underline{y}\right)$
$\overline{z} = \max\left(\underline{x} \hat{/} \underline{y}, \overline{x} \hat{/} \overline{y}, \underline{x} \hat{/} \overline{y}, \overline{x} \hat{/} \underline{y}\right)$
end

Note that the symbols $\check{\circ}$ and $\hat{\circ}$ indicate that any binary operation $\circ \in \{+, -, \times, \div\}$ was executed through round downward and through round upward, respectively.

§ 2.2. Implementation 2 : Interval Arithmetic of inf-sup Type using Only Round-to-Nearest

This section describes calculation methods using the round-to-nearest mode that can return results at the same precision level as those using the round upward or round downward mode for any inf-sup type intervals $[x]$ and $[y]$. The calculation methods regarding inf-sup type intervals in the expressions **Algorithm 4** – **Algorithm 7** reveal that only an error caused by at most one floating-point arithmetic operation is mixed into any result of inf-sup type interval operations. Due to this fact, the precision comparable to that of interval operations using round upward or round downward can be obtained through round-to-nearest without changing it to either of those rounding modes by identifying that error caused by one floating-point arithmetic operation and judging its sign.

Algorithm 8 Solution interval for one floating-point arithmetic operation

Consider two floating-point numbers $a, b \in \mathbb{F}$. Assuming that floating-point number c is given in (2.5) and that e is given in (2.6) for any binary operation $\circ \in \{+, -, \times, \div\}$,

$$c = fl\,(a \circ b)\,, \tag{2.5}$$

$$e = a \circ b - c, \tag{2.6}$$

the following holds:

$$a \circ b \in [(e \geqq 0) * c + (e < 0) * \mathrm{pred}\,(c)\,,\ (e \leqq 0) * c + (e > 0) * \mathrm{succ}\,(c)]\,. \tag{2.7}$$

For the succ and pred functions, see Appendix. Also, the judging function such as $e \geqq 0$ is a function that returns 1 if the judgment result is true and returns 0 otherwise.

Algorithm 9 – **Algorithm 12** show interval operations that use only the round-to-nearest mode. Note that in **Algorithm 9**–**Algorithm 12**, the following (2.8) holds true for any binary operation $\circ \in \{+, -, \times, \div\}$ and any inf-sup type intervals $[x]$ and $[y]$:

$$[z] \supseteq [x] \circ [y]. \tag{2.8}$$

Note that $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$ indicates that the operation was executed with round-to-nearest for any binary operation $\circ \in \{+, -, \times, \div\}$. Furthermore, to describe algorithms using TwoSum and TwoProduct below, we use MATLAB-like programming constructs. See Appendix A.4 about TwoSum and TwoProduct algorithms.

Algorithm 9 Interval addition using round-to-nearest

```
function z = Addition(x, y)
    [z, e1] = TwoSum(x, y);
    [z̄, e2] = TwoSum(x̄, ȳ);
    if e1 < 0, z = pred(z); end;
    if e2 > 0, z̄ = succ(z̄); end;
end
```

Algorithm 10 Interval subtraction using round-to-nearest

```
function z = Subtraction(x, y)
    [z, e1] = TwoSum(x, −ȳ);
    [z̄, e2] = TwoSum(x̄, −y);
    if e1 < 0, z = pred(z); end;
    if e2 > 0, z̄ = succ(z̄); end;
end
```

Algorithm 11 Interval multiplication using round-to-nearest

function $z = \text{Multiplication}\,(x, y)$
$[a, b] = \text{MulLo}(\underline{x}, \overline{x}, \underline{y}, \overline{y})$;
$[c, d] = \text{MulUp}(\underline{x}, \overline{x}, \underline{y}, \overline{y})$;
$[\underline{z}, e_1] = \text{TwoProduct}\,(a, b)$;
$[\overline{z}, e_2] = \text{TwoProduct}\,(c, d)$;
if $e_1 < 0,\ \underline{z} = \text{pred}\,(\underline{z})$; end;
if $e_2 > 0,\ \overline{z} = \text{succ}\,(\overline{z})$; end;
end

Algorithm 12 Interval division using round-to-nearest

function $z = \text{Division}\,(x, y)$
$[a, b] = \text{DivLo}(\underline{x}, \overline{x}, \underline{y}, \overline{y})$;
$[c, d] = \text{DivUp}(\underline{x}, \overline{x}, \underline{y}, \overline{y})$;
$\underline{z} = a \oslash b$;
$\overline{z} = c \oslash d$;
$[s_1, s_2] = \text{TwoProduct}\,(\underline{z}, b)$;
$[t_1, t_2] = \text{TwoProduct}\,(\overline{z}, d)$;
$e_1 = (s_1 \ominus a) \oplus s_2$;
$e_2 = (t_1 \ominus c) \oplus t_2$;
if $e_1 > 0,\ \underline{z} = \text{pred}\,(\underline{z})$; end;
if $e_2 < 0,\ \overline{z} = \text{succ}\,(\overline{z})$; end;
end

Note that MulLo and MulUp are functions that select two elements making up the infimum and the supremum of the expression (1.9) among $\underline{x}, \overline{x}, \underline{y}, \overline{y}$, and DivLo and DivUp are functions that select two elements making up the infimum and the supremum of the expression (1.10) among $\underline{x}, \overline{x}, \underline{y}, \overline{y}$.

§ 2.3. Implementation 3 : Fast Interval Arithmetic of inf-sup Type using Only Round-to-Nearest

This section describes rapid interval operation methods using only the round-to-nearest mode for any inf-sup type intervals $[x]$ and $[y]$. As observing the expression (2.7) reveals that the increase of the interval width is 1 bit at most, it is possible to remove this branch if the rapidity of calculation must be emphasized.

Algorithm 13 Solution interval for one floating-point arithmetic operation

Take any two floating-point numbers $a, b \in \mathbb{F}$. Assuming that a floating-point number c satisfies the following (2.9) for any binary operation $\circ \in \{+, -, \times, \div\}$,

$$c = fl\,(a \circ b)\,, \tag{2.9}$$

the following holds:

$$a \circ b \subseteq [\mathrm{pred}\,(c)\,,\ \mathrm{succ}\,(c)]\,. \tag{2.10}$$

Algorithm 14–**Algorithm 17** show interval operation methods that use only the round-to-nearest mode. Note that in **Algorithm 14**–**Algorithm 17**, the following holds true for any binary operation $\circ \in \{+, -, \times, \div\}$ and any inf-sup type intervals $[x]$ and $[y]$:

$$[z] \supseteq [x] \circ [y]. \tag{2.11}$$

Algorithm 14 Fast Interval addition using round-to-nearest

function $z = \mathrm{Addition}\,(x, y)$
$\underline{z} = \mathrm{pred}\,(\underline{x} \oplus \underline{y})\,;$
$\overline{z} = \mathrm{succ}\,(\overline{x} \oplus \overline{y})\,;$
end

Algorithm 15 Fast Interval subtraction using round-to-nearest

function $z = \mathrm{Subtraction}\,(x, y)$
$\underline{z} = \mathrm{pred}\,(\underline{x} \ominus \overline{y})\,;$
$\overline{z} = \mathrm{succ}\,(\overline{x} \ominus \underline{y})\,;$
end

Algorithm 16 Fast Interval multiplication using round-to-nearest

function $z = \mathrm{Multiplication}\,(x, y)$
$[a, b] = \mathrm{MulLo}(\underline{x}, \overline{x}, \underline{y}, \overline{y});$
$[c, d] = \mathrm{MulUp}(\underline{x}, \overline{x}, \underline{y}, \overline{y});$
$\underline{z} = \mathrm{pred}\,(a \otimes b)\,;$
$\overline{z} = \mathrm{succ}\,(a \otimes b)\,;$
end

Algorithm 17 Fast Interval division using round-to-nearest

function $z = \mathrm{Division}\,(x, y)$
$[a, b] = \mathrm{DivLo}(\underline{x}, \overline{x}, \underline{y}, \overline{y});$
$[c, d] = \mathrm{DivUp}(\underline{x}, \overline{x}, \underline{y}, \overline{y});$
$\underline{z} = \mathrm{pred}\,(a \oslash b)\,;$
$\overline{z} = \mathrm{succ}\,(c \oslash d)\,;$
end

§ 3. Interval Arithmetic for mid-rad Type with Double-Precision Floating-Point Numbers

Rewriting the expression (1.6) for the four arithmetic operations, identification of the inf-sup type intervals with the mid-rad type intervals yields those operations for mid-rad type intervals as described in **Algorithm 18**.

Algorithm 18 Four arithmetic operations for any intervals $\langle x \rangle,\ \langle y \rangle \in \mathbb{IR}$

$$\langle x \rangle + \langle y \rangle = \langle x_c + y_c,\ x_r + y_r \rangle \tag{3.1}$$

$$\langle x \rangle - \langle y \rangle = \langle x_c - y_c,\ x_r + y_r \rangle \tag{3.2}$$

$$\langle x \rangle \times \langle y \rangle = \langle m_c, m_r \rangle \tag{3.3}$$

$$m_c = x_c y_c + \operatorname{sign}(x_c y_c) \min\left(|x_c|\, y_r, x_r\, |y_c|, x_r y_r\right) \tag{3.4}$$

$$m_r = |x_c|\, y_r + x_r\, |y_c| + x_r y_r - \min\left(|x_c|\, y_r, x_r\, |y_c|, x_r y_r\right) \tag{3.5}$$

$$\langle x \rangle\ /\, \langle y \rangle = \frac{\langle x \rangle \times \langle y \rangle}{\underline{y}\overline{y}} = \frac{\langle x \rangle \times \langle y \rangle}{y_c^2 - y_r^2} \qquad (0 \notin \langle y \rangle) \tag{3.6}$$

The case calculations can be also used in multiplication. **Algorithm 19** shows such calculations.

Algorithm 19 Multiplication method for any real intervals $\langle x \rangle,\ \langle y \rangle \in \mathbb{IR}$

	$0 \notin \langle x \rangle$	$0 \in \langle x \rangle$
$0 \notin \langle y \rangle$	I_1	I_2
$0 \in \langle y \rangle$	I_3	I_4

$$I_1 = \langle x_c y_c + \operatorname{sign}(x_c y_c)\, x_r y_r,\ |x_c|\, y_r + x_r\, |y_c| \rangle \tag{3.7}$$

$$I_2 = \langle x_c\, (y_c + \operatorname{sign}(y_c)\, y_r),\ x_r\, (|y_c| + y_r) \rangle \tag{3.8}$$

$$I_3 = \langle (x_c + \operatorname{sign}(x_c)\, x_r)\, y_c,\ (|x_c| + x_r)\, y_r \rangle \tag{3.9}$$

$$I_4 = \langle x_c y_c + \operatorname{sign}(x_c y_c) \min\left(|x_c|\, y_r, x_r\, |y_c|\right),\ \max\left(|x_c|\, y_r, x_r\, |y_c|\right) + x_r y_r \rangle \tag{3.10}$$

As for the interval multiplication, the following calculation method without branches (**Algorithm 20**) has been widely known when the radius of the interval is comparatively small relative to the center:

Algorithm 20 Four arithmetic operations for any intervals $\langle x\rangle\,,\ \langle y\rangle \in \mathbb{IR}$

$$\langle x\rangle \times \langle y\rangle \subseteq \langle x_c y_c,\ |x_c|\, y_r + x_r\, |y_c| + x_r y_r\rangle \tag{3.11}$$

The right-hand side of **Algorithm 20** is overestimated; it is known that the width is at most approximately 1.5 times the interval width that is obtained with exact calculation ([1]).

Let $\mathbb{IM}$ be a set of mid-rad intervals that have floating-point numbers as their elements:

$$\mathbb{IM} = \{\langle x\rangle \in \mathbb{IR}\,|x_c,\ x_r \in \mathbb{F}\,\}. \tag{3.12}$$

Given any two intervals $\langle x\rangle\,, \langle y\rangle \in \mathbb{IM}$, the binary operation on them shall be defined as follows:

$$\langle x\rangle \circ \langle y\rangle \supseteq \{x \circ y\,|\ \text{for}\ \forall x \in \langle x\rangle\,,\ \forall y \in \langle y\rangle\,\}. \tag{3.13}$$

Similarly, the unary operation $f\,(x)$ shall be defined as follows:

$$f\,(\langle x\rangle) \supseteq \{f\,(x)\,|\ \text{for}\ \forall x \in \langle x\rangle\,\}. \tag{3.14}$$

These operations are referred to as mid-rad type interval operations.

§ 3.1. Implementation 1 : Interval Arithmetic of mid-rad Type using Changed Rounding

IEEE Standard 754 defines the rounding modes for operations. Using the round upward or round downward mode enables the four arithmetic operations for the mid-rad type intervals to be executed similarly to those for the inf-sup type intervals.

The four arithmetic operations for the mid-rad type mechanical interval operations using the upward or round downward mode according to **Algorithm 18** can be written as shown in **Algorithm 21**–**Algorithm 24**. Note that, in **Algorithm 21**–**Algorithm 24**, the following holds true for any binary operation $\circ \in \{+, -, \times, \div\}$ and any mid-rad type intervals $\langle x\rangle$ and $\langle y\rangle$:

$$\langle z\rangle \supseteq \langle x\rangle \circ \langle y\rangle\,. \tag{3.15}$$

Algorithm 21 Interval addition using changed rounding modes

function $z = \mathrm{Addition}(x, y)$
$z_c = x_c \check{+} y_c;$
$e_c = (x_c \hat{+} y_c) \hat{-} z_c;$
$z_r = (x_r \hat{+} y_r) \hat{+} e_c;$
end

Algorithm 22 Interval subtraction using changed rounding modes

function $z = \mathrm{Subtraction}(x, y)$
$z_c = x_c \check{-} y_c;$
$e_c = (x_c \hat{-} y_c) \hat{-} z_c;$
$z_r = (x_r \hat{+} y_r) \hat{+} e_c;$
end

Algorithm 23 Interval multiplication using changed rounding modes

function $z = \mathrm{Multiplication}(x, y)$
$c_1 = \min(|x_c| \check{\times} y_r, x_r \check{\times} |y_c|, x_r \check{\times} y_r);$
$c_2 = |x_c| \check{\times} |y_c| \check{+} c_1;$
$c_3 = |x_c| \hat{\times} |y_c| \hat{+} c_1;$
$z_c = (c_3 \hat{+} c_2) \hat{/} 2;$
$e_c = z_c \hat{-} c_2;$
$z_r = |x_c| \hat{\times} x_r \hat{+} x_r \hat{\times} |y_c| \hat{+} x_r \hat{\times} y_r \hat{-} c_1 \hat{+} e_c;$
$z_c = \mathrm{sign}(x_c \hat{\times} y_c)\, z_c;$
end

Algorithm 24 Interval division using changed rounding modes

function $z = \mathrm{Division}(x, y)$
$t_1 = (|y_c| \hat{-} y_r) \hat{\times} (|y_c| \hat{+} y_r);$
$t_2 = (|y_c| \check{-} y_r) \check{\times} (|y_c| \check{+} y_c);$
$c_1 = \min(|x_c| \check{\times} x_r, x_r \check{\times} |y_c|, x_r \check{\times} y_r);$
$c_2 = (|x_c| \check{\times} |y_c| \check{+} c_1) \check{/} t_1;$
$c_3 = (|x_c| \hat{\times} |y_c| \hat{+} c_1) \hat{/} t_2;$
$z_c = (c_3 \hat{+} c_2) \hat{/} 2;$
$e_c = z_c \hat{-} c_2;$
$z_r = (|x_c| \hat{\times} x_r \hat{+} x_r \hat{\times} |y_c| \hat{+} x_r \hat{\times} y_r \hat{-} c_1) \hat{/} t_2 \hat{+} e_c;$
$z_c = \mathrm{sign}(x_c \hat{\times} y_c)\, z_c;$
end

Adopting **Algorithm 20** for the multiplication method enables the above to be written as described in **Algorithm 25**–**Algorithm 26**.

Algorithm 25 Interval multiplication using changed rounding modes

function $z = \text{Multiplication}(x, y)$
$z_c = x_c \check{\times} y_c$;
$e_c = x_c \hat{\times} y_c \hat{-} z_c$;
$z_r = \left(\left(|x_c| \hat{+} x_r\right) \hat{\times} y_r\right) \hat{+} \left(x_r \hat{\times} |y_c|\right) \hat{+} e_c$;
end

Algorithm 26 Interval division using changed rounding modes

function $z = \text{Division}(x, y)$
$t_1 = \left(|y_c| \hat{-} y_r\right) \hat{\times} \left(|y_c| \hat{+} y_r\right)$;
$t_2 = \left(|y_c| \check{-} y_r\right) \check{\times} \left(|y_c| \check{+} y_c\right)$;
$c_1 = |x_c| \check{\times} |y_c| \check{/} t_1$;
$c_2 = |x_c| \hat{\times} |y_c| \hat{/} t_2$;
$z_c = \left(c_2 \hat{+} c_1\right) \hat{/} 2$;
$e_c = z_c \hat{-} c_1$;
$z_r = \left(|x_c| \hat{\times} y_r \hat{+} x_r \hat{\times} |y_c| \hat{+} x_r \hat{\times} y_r\right) \hat{/} t_2 \hat{+} e_c$;
$z_c = \text{sign}\left(x_c \hat{\times} y_c\right) z_c$;
end

§ 3.2. Implementation 2 : Interval Arithmetic of mid-rad Type using Only Round-to-Nearest

This section describes interval arithmetic using the round-to-nearest mode that can return results at the same precision level as those using the round upward or round downward mode for any mid-rad type intervals $\langle x \rangle$ and $\langle y \rangle$.

First, an error evaluation which is used in algorithms of this section is shown. See [8] for more details about FastTwoSum and SumL.

Algorithm 27 Error evaluation with SumL2

Let p be a vector whose element is n (where $n \leqq 5$). Let two floating point numbers q_1 and q_2 be the following:

$$[q_1, q_2] = \text{FastTwoSum}\left(\text{SumL}\left(p, 2\right)\right). \tag{3.16}$$

When q_1 and q_2 satisfy the following

$$|q_2| \geqq 2^{-100} |q_1|, \tag{3.17}$$

the following holds:

$$\sum_{i=1}^{n} p_i \leqq (q_2 \leqq 0) * q_1 + (q_2 > 0) * \text{succ}\left(q_1\right). \tag{3.18}$$

Algorithm 27 can be derived from error analysis with SumL.

Algorithm 28 – **Algorithm 30** show interval operation methods that use only the round-to-nearest mode. As the division is too complex, however, it is not described in this section. Note that, in **Algorithm 28**–**Algorithm 30**, the following holds true for any binary operation $\circ \in \{+, -, \times\}$ and any mid-rad type intervals $\langle x \rangle$ and $\langle y \rangle$:

$$\langle z \rangle \supseteq \langle x \rangle \circ \langle y \rangle \,. \tag{3.19}$$

Algorithm 28 Interval addition using changed rounding modes

```
function z = Addition(x, y)
    [z_c, e_c] = TwoSum(x_c, y_c);
    [z_r, e_r] = FastTwoSum(SumL([x_r, y_r, |e_c|], 2));
    if |e_r| > 2^-100 z_r && e_r < 0, return; end;
    z_r = succ(z_r);
end
```

Algorithm 29 Interval subtraction using changed rounding modes

```
function z = Subtraction(x, y)
    [z_c, e_c] = TwoSum(x_c, -y_c);
    [z_r, e_r] = FastTwoSum(SumL([x_r, y_r, |e_c|], 2));
    if |e_r| > 2^-100 z_r && e_r < 0, return; end;
    z_r = succ(z_r);
end
```

Algorithm 30 Interval multiplication using changed rounding modes

```
function z = Multiplication(x, y)
    [c_1, c_2] = TwoProduct(x_c, y_c);
    [c_3, c_4] = TwoProduct(MulMin(x_c, x_r, y_c, y_r));
    [r_1, r_2] = TwoProduct(MulMax(x_c, x_r, y_c, y_r));
    [r_3, r_4] = TwoProduct(MulMax2(x_c, x_r, y_c, y_r));
    [z_c, e_c] = SumL([c_1, c_2, sign(c_1) c_3, sign(c_1) c_4], 2);
    e_c = succ(|e_c| + 2^-100 z_c);
    [z_r, e_r] = FastTwoSum(SumL([r_1, r_2, r_3, r_4, e_c], 2));
    if |e_r| > 2^-100 z_r && e_r < 0, return; end;
    z_r = succ(z_r);
end
```

§ 3.3. Implementation 3 : Fast Interval Arithmetic of mid-rad Type using Only Round-to-Nearest

This section describes rapid interval operation methods using only round-to-nearest for any mid-rad type intervals $\langle x \rangle$ and $\langle y \rangle$.

The four arithmetic operations for the mid-rad type fast interval operations using the round-to-nearest mode can be written as shown in **Algorithm 31**–**Algorithm 33**. Note that in **Algorithm 31**–**Algorithm 33**, the following holds true for any binary operation $\circ \in \{+, -, \times, \div\}$ and any mid-rad type operations $\langle x \rangle$ and $\langle y \rangle$:

$$\langle z \rangle \supseteq \langle x \rangle \circ \langle y \rangle \,. \tag{3.20}$$

Algorithm 31 Interval addition using round-to-nearest

function $z = \mathrm{Addition}\,(x, y)$
$[z_c, e_c] = \mathrm{TwoSum}(x_c, y_c);$
$z_r = \mathrm{succ2}\,(x_r \oplus y_r \oplus |e_c|)\,;$
end

Algorithm 32 Interval subtraction using round-to-nearest

function $z = \mathrm{Subtraction}\,(x, y)$
$[z_c, e_c] = \mathrm{TwoSum}(x_c, -y_c);$
$z_r = \mathrm{succ2}\,(x_r \oplus y_r \oplus |e_c|)\,;$
end

Algorithm 33 Interval multiplication using round-to-nearest

function $z = \mathrm{Multiplication}\,(x, y)$
$[z_c, e_c] = \mathrm{TwoProduct}\,(x_c, y_c)\,;$
$z_r = \mathrm{succ6}\,(|x_c| \otimes y_r \oplus x_r \otimes |y_c| \oplus x_r \otimes y_r \oplus |e_c|)\,;$
end

Note that the succ$n$ function is a function that executes succ n times.

§ 4. Interval Arithmetic for Matrices

A matrix every element of which is an interval is called an interval matrix. When any matrix operation is executed, it is required that additions, subtractions, multiplications, and divisions between any intervals belonging to the matrix are rapidly executed. Particularly, any calculation method for which a fast linear algebra library, such as BLAS used in approximations, can be used directly is desirable because the execution time is shortened. This section only describes the outline of matrix interval operations.

As described previously, in order to calculate a tight interval, some branches are required for the interval multiplication. However, when considering the matrix multiplication, the execution time would be extremely longer if the branches occurred for each multiplication of scalar numbers. In order to overcome this problem, this section describes calculation techniques that execute the interval multiplication rapidly by using a fast linear algebra library.

The following algorithm **Algorithm 34** has been widely used:

Algorithm 34 Addition, subtraction, and multiplication methods on any interval matrices $\langle x \rangle\,,\ \langle y \rangle \in \mathbb{IR}^{n \times n}$

$$\langle x \rangle + \langle y \rangle = \langle x_c + y_c,\ x_r + y_r \rangle \tag{4.1}$$

$$\langle x \rangle - \langle y \rangle = \langle x_c - y_c,\ x_r + y_r \rangle \tag{4.2}$$

$$\langle x \rangle \times \langle y \rangle \subseteq \langle x_c y_c,\ |x_c|\, y_r + x_r\, |y_c| + x_r y_r \rangle \tag{4.3}$$

Expression (4.3) and **Algorithm 20** are identical except that the former includes scalar multiplications, but the latter includes matrix multiplications. This calculation technique does not include any branches, and it is known that the interval increases such that the maximum interval increase ratio is 1.5 at most with an average ratio of 1.18 approximately ([4]).

The next **Algorithm 35** is a high-precision calculation method for interval matrix products, which uses a fast linear algebra library by executing all branches in advance.

Algorithm 35 Calculation Method for Interval Matrix Products

Let two matrices β and γ be the following (4.4) and (4.5) for any interval matrices $\langle x\rangle, \langle y\rangle \in \mathbb{IR}^{n\times n}$:

$$\beta = \{\mathrm{sign}(x_c)\min(|x_c|, x_r)\} \times \{\mathrm{sign}(y_c)\min(|y_c|, y_r)\} \tag{4.4}$$

$$\gamma = \min(|x_c|, x_r)\min(|y_c|, y_r). \tag{4.5}$$

Calculate the interval product as follows:

$$\langle x\rangle \times \langle y\rangle \subseteq < x_c y_c + \beta, |x_c| x_r + |y_c| x_r + x_r y_r - \gamma > . \tag{4.6}$$

It is known that the maximum ratio of interval increase for **Algorithm 35** is approximately 1.18 at most ([4]).

The following comparison table lists the features between the interval matrix product calculation methods:

Table 1. Comparison between mid-rad Type Interval Matrix Product Methods.

	Interval Enlargement		Fast Linear Algebra Library
	Max.	Ave.	
(3.3)	1.0 times	1.0 times	Not Applicable
(4.3)	1.5 times	1.18 times	Applicable
(4.6)	1.18 times	1.01 times	Applicable

§ 5. General Notes on Implementation

The interval arithmetic is a calculation method that puts an emphasis on calculations free from any errors, considering not only real number calculations, but also rounding errors in computers. For this reason, it is not allowed that the method includes even one-bit miscalculation in executing calculations. This article requires that those interval calculations conform to IEEE Standard 754 and introduces several algorithms based on that assumption. Actual computers involve a complex mixture of several factors, including the following:

- CPU settings,
- Compiler settings,
- Fast linear algebra library settings.

It is required to configure the intended settings for all of them.

In the CPU settings, settings for change from the default rounding are particularly important. In the past, x87 FPU operational units have been widely used[1] , and the faster operation has recently been accomplished with SSE, while changing the default rounding has become further complicated. As some environments use both the operational units, the default round mode must be changed for both in such environments.

In addition to this, the compiler settings are more complex. Let us quote one paragraph from "Computer Organization and Design" by D. Patterson & J. Hennessy ([5]).

> "Unfortunately, the compiler-writing community was not represented adequately in the wrangling, and some of the features didn't balance language and compiler issues against other points. That community has been slow to make IEEE Standard 754's unusual features available to the applications programmer. Humane exception handling is one such unusual feature; directed rounding another. Without compiler support, these features have atrophied."

For these issues, this chapter describes an implementation of a safe interval operation using the gcc complier capable of compiling SIMD intrinsic instructions in a 64-bit machine that supports SSE2.

§ 5.1. SIMD intrinsic Instructions

The SIMD (acronym of Single Instruction Multiple Data) operation refers to a set of instructions, each instruction of which can process multiple pieces of data. The operation has recently been implemented even on a wide range of general-purpose CPUs for PCs. SSE2 , which was implemented on Intel Pentium4 in November 2000, is a SIMD instruction set that can process two double-precision floating-point number arithmetic operations at once using 128-bit registers. The intrinsic instruction is a group of header files to implement SSE2 on the C language. To use SSE2, the following three header files must be included:

```
#include <emmintrin.h>   // MMX-SSE2//
#include <xmmintrin.h>   // MMX-SSE//
#include <mmintrin.h>    // MMX//
```

When using these header files, gcc provides the compile option: `-msse2 -mfpmath=sse`.

[1]The existing x87 FPU handles 80-bit extended double-precision floating-point numbers and can internally execute computations with higher precision than SSE2 handling 128 (64×2)-bit SSE2; as it has a possibility that the precision may not be validated due to double rounding caused by using 80 bits, operations with SSE2 should be regarded as safe for now.

§ 5.2. Changing Rounding Mode

This section describes the declarations for variables, macros, and functions useful to change the rounding mode in SSE2.

```
unsigned int _CtrlWordSSE       __attribute__ ((aligned (16)));
unsigned int _RoundDownSSE      __attribute__ ((aligned (16)));
unsigned int _RoundUpSSE        __attribute__ ((aligned (16)));

#define WriteCtrlWordSSE()      _mm_setcsr(_CtrlWordSSE);
#define RoundDownSSE()          _mm_setcsr(_RoundDownSSE);
#define RoundUpSSE()            _mm_setcsr(_RoundUpSSE);

inline void ReadCtrlWordSSE(){
        _CtrlWordSSE  = _mm_getcsr();
        _RoundDownSSE = ((~0x00006000) & _CtrlWordSSE) | 0x00002000;
        _RoundUpSSE   = ((~0x00006000) & _CtrlWordSSE) | 0x00004000;
}
```

When changing the rounding mode, one must save the information on the floating-point arithmetic flags in a program before the location where one wants to change the rounding mode within the program. To save the information, include `ReadCtrlWordSSE()`, and to restore it, include `WriteCtrlWordSSE()`. To set the rounding mode to round downward, include `RoundDownSSE()`, and to set the rounding mode to round upward, include `RoundUpSSE()`.

To confirm whether the changed rounding mode functions properly, the following method has been widely known: When conducting $1 + e$ and $1 - e$ on the floating-pointing number $e = 2^{-60}$, one gets results as listed in Table 2 so that one may check which rounding mode functions now.

Table 2. Fast Rounding Mode Confirmation Method

	$1 + e$	$1 - e$
Round to nearest	1	1
Round downward	1	pred (1)
Round upward	succ (1)	1

As an example, **Algorithm 36** describes the major part of the inf-sup type interval addition (**Algorithm 4**) using a changed rounding mode.

Algorithm 36 Main part of **Algorithm 4**

```
ReadCtrlWordSSE();
RoundDownSSE();
z.lower() = x.lower() + y.lower();
RoundUpSSE();
z.upper() = x.upper() + y.upper();
WriteCtrlWordSSE();
```

§ 5.3. Interval Arithmetic using Only Round-To-Nearest

Error-free transformations such as TwoProduct are useful to obtain results with high precision, but they may not function properly, depending on the degree of compiler optimization. To resolve this problem, one can rewrite TwoProduct with SSE2.

```
inline void TwoProduct(double a, double b, double *x, double *y){
        __m128d ab ={a,b};
        __m128d fac = {134217729, 134217729};     %2^27+1
        __m128d ab1, ab2;
        __m128d p1, p2;
        __m128d r0, r1, r2;
        __m128d s1, s2, s3, s4;
        __m128d u1, u2, u3, u4;
        double u[2] __attribute__ ((aligned (16)));

        *x = a*b;

        p1 = _mm_mul_pd(ab,fac);
        p2 = _mm_sub_pd(p1,ab);
        ab1= _mm_sub_pd(p1,p2);
        ab2= _mm_sub_pd(ab,ab1);

        r0 = _mm_shuffle_pd(ab2,ab2,1);
        r1 = _mm_shuffle_pd(ab1,ab2,0);
        r2 = _mm_shuffle_pd(ab1,ab2,3);

        s2 = _mm_mul_pd(ab1,r0);
        s1 = _mm_mul_pd(r1,r2);
```

```
        s3 = _mm_shuffle_pd(s2,s2,1);
        s4 = _mm_shuffle_pd(s1,s1,1);

        x0 = _mm_set1_pd(*x);
        u1 = _mm_sub_pd(x0,s1);
        u2 = _mm_sub_pd(u1,s2);
        u3 = _mm_sub_pd(u2,s3);
        u4 = _mm_sub_pd(s4,u3);

        _mm_store_pd(u,u4);
        *y = u[0];
}
```

Even when one compiles a program with the gcc compile option

```
-O3 -msse2 -mfpmath=sse -ffast-math
```

the correct result from executing TwoProduct will be output. One must note, however, that as the `-ffast-math` option is a speeding-up option, the program may not conform to IEEE Standard 754.

§ 5.4. Changing Rounding Mode for x87 FPU

In some numerical computation environments, one wishes to change even the rounding mode for x87 FPU. If that is the case, one can change it by using the following source.

```
unsigned short int _CtrlWord    __attribute__ ((aligned (16)));
unsigned short int _RoundDown   __attribute__ ((aligned (16)));
unsigned short int _RoundUp     __attribute__ ((aligned (16)));

#define WriteCtrlWord() __asm__ __volatile__ ("fldcw %0" : : "m"(_CtrlWord));
#define RoundDown()     __asm__ __volatile__ ("fldcw %0" : : "m"(_RoundDown));
#define RoundUp()       __asm__ __volatile__ ("fldcw %0" : : "m"(_RoundUp));

inline void ReadCtrlWord(){
        __asm__ __volatile__ ("fnstcw %0" : "=m"(_CtrlWord));
        _RoundDown = ((~0x0c00) & _CtrlWord) | 0x0400;
        _RoundUp   = ((~0x0c00) & _CtrlWord) | 0x0800;
}
```

When changing the rounding mode, one must save the information on the floating-point arithmetic flags in a program before the location where one wants to change the

rounding mode within the program. To save the information, include `ReadCtrlWord()`, and to restore it, include `WriteCtrlWord()`.

To set the rounding mode to round downward, include `RoundDown()`, and to set the rounding mode to round upward, include `RoundUp()`.

Acknowledgement

We would like to express gratitude to Professor Masahide Kashiwagi, Waseda University who gave us many pieces of advice when preparing this article, and we also heartily wish to thank the anonymous referees for their thorough reading and most valuable comments.

This work was supported by JSPS KAKENHI Grant Number 24700015.

References

[1] S. Oishi: Verified Numerical Computations, Corona Publishing Co.,Ltd., 1999. (in Japanese)
[2] A. Neumaier: Interval Methods for Systems of Equations, Cambridge University Press, 1990.
[3] S. M. Rump: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica, 19:287–449, 2010.
[4] S.M. Rump : Fast Interval Matrix Multiplication. Numerical Algorithms, 61(1):1–34, 2012.
[5] D. Patterson & J. Hennessy: Computer Organization and Design. Morgan Kaufmann, 1997.
[6] D. E. Knuth: The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, Reading, Massachusetts, 1969.
[7] T. J. Dekker: A floating-point technique for extending the available precision, Numer. Math., 18 (1971), pp. 224–242.
[8] T. Ogita, S. M. Rump, S. Oishi: Accurate sum and dot product, SIAM J. Sci. Comput., 26:6 (2005), pp. 1955–1988.

§ Appendix A. Basic Knowledge of Floating-point Numbers and Their Operations

This article has been prepared on the basis of floating point numbers in compliance with IEEE Standard 754, which are used in most numerical calculations with current computers. This section describes IEEE Standard 754 as a basis of numerical calculation methods.

Appendix A.1. Floating-Point Numbers

A floating-point number system based on IEEE Standard 754 is defined by a set of floating-point numbers and operations on it. IEEE Standard 754 specifies and provides the following floating-point numbers: normalized numbers, zeros, denormalized numbers, and floating-point number exceptions (Infinities and NaNs).

Normalized numbers

For d_i that takes either 0 or 1, a number that can be written as follows is referred to as a normalized number:

$$a = \pm \left(\frac{1}{2^0} + \frac{d_1}{2^1} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \cdots + \frac{d_{N-1}}{2^{N-1}} \right) \times 2^e.$$

Assuming that $e_{\min}$ is a negative integer and $e_{\max}$ is a positive integer, e is an integer that satisfies $e_{\min} \leqq e \leqq e_{\max}$. The part defined by

$$m := \pm \left(\frac{1}{2^0} + \frac{d_1}{2^1} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \cdots + \frac{d_{N-1}}{2^{N-1}} \right),$$

is referred to as signed mantissa, and e is referred to as exponent. The exponent e can be also expressed in binary. Generally, floating-point systems could be those with single precision (4 byte = 32 bit), with double precision (8 byte = 64 bit), and with extended double precision (16 byte = 128 bit), and they are represented with following floating-point systems:

Single Precison	$N = 24, \quad -126 \leqq e \leqq 127$
Double Precision	$N = 53, \quad -1022 \leqq e \leqq 1023$
Extended Double Precision	$N = 64, -16382 \leqq e \leqq 16383$

When considering floating-point numbers, this article shall refer to double-precision floating-point numbers that conform to IEEE Standard 754 unless otherwise specified.

The maximum of the absolute values of normalized numbers is

$$x_{\max} = \left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{N-1}} \right) \times 2^{e_{\max}},$$

and the minimum is $x_{\min} = 2^{e_{\min}}$. When $|x| > x_{\max}$ is true, overflow is said to occur. As the signed mantissa part consists of 53 bits in any double-precision floating-point number

$$2^{-53} = 1.110223 \cdots \times 10^{-16},$$

the double-precision floating-point number has a precision of approximately 16 digits in decimal.

Zeros

Zeros are normalized and expressed as the following:

$$+\left(\frac{0}{2^0} + \frac{0}{2^1} + \frac{0}{2^2} + \frac{0}{2^3} + \cdots + \frac{0}{2^{N-1}}\right) \times 2^{e_{\min}}.$$

Denormalized numbers

To express any floating-point number whose mantissa part is smaller than 1 and exponent part is $e_{\min}$, IEEE Standard 754 stops defaulting the first digit to 1, and allows a number of 0 to be set in the place. This number is called a denormalized number. When any number falls under the range of denormalized numbers, gradual underflow is said to occur. The smallest positive number among denormalized numbers is

$$2^{-1074} = 4.940656 \cdots \times 10^{-324}.$$

When any number becomes smaller than or equal to this, underflow is said to occur.

Floating-point number exceptions

In addition to the above, IEEE Standard 754 provides the following special numbers:

- Inf (Infinity) is obtained as a result of an overflow or zero division.
- NaN (Not a Number) is obtained as a resuof an illegal operation such as

$$\sqrt{-5},\ \frac{\infty}{\infty},\ +\infty,\ +(-\infty)\,.$$

- ± 0 is obtained as a result of a division with $\pm\infty$ after an underflow occurs.

Appendix A.2. Rounding Modes and Four Arithmetic Operations

Rounding

IEEE Standard 754 defines any floating-point number operation with rounding. Let c be any real number.

- Round upward $\triangle : \mathbb{R} \to \mathbb{F}$
  Any number will be rounded to the smallest floating-point number larger than or equal to c.

- Round downward $\triangledown : \mathbb{R} \to \mathbb{F}$
  Any number will be rounded to the largest floating-point number smaller than or equal to c.

- Round to nearest $\bigcirc$ or $fl(\cdots) : \mathbb{R} \to \mathbb{F}$
  Any number will be rounded to the nearest floating-point number to c. If two nearest numbers are found, the number will be rounded to the one the last bit of whose signed mantissa part is even (this is called even number rounding).

- Round toward 0
  Any number will be rounded to the floating-point number that is nearest to and no greater than c in magnitude.

Four arithmetic operations

IEEE Standard 754 specifies that the result of any four arithmetic operations on floating-point numbers must match the one derived from executing the specified rounding on the result of the corresponding mathematically correct operation using real numbers. The result of square root as a unary operation must satisfy the similar property. It means that the operation rules, the accuracy of whose calculation results is validated in numerical calculation, are only four arithmetic operations and square root. What one should note is that other operations such as exponential function and trigonometric functions have not been standardized with any standards such as IEEE Standard 754. This means that another ingenious attempt must be made to obtain the values of these elementary functions with validated accuracy.

In addition, decimal-to-binary conversions as well as the opposite conversions are often required in numerical calculations. One must note that any decimal integer can be converted to the binary without any error and that an error occurs when converting any decimal fraction to the binary.

Appendix A.3. Features of Floating-Point Numbers and Their Operations

pred • succ

The function pred and succ shall be defined as follows:

$$\text{pred}(r) := \max\{f \in \mathbb{F} \ : \ f < r\} \tag{A.1}$$

$$\text{succ}(r) := \min\{f \in \mathbb{F} \ : \ r < f\}. \tag{A.2}$$

Next, **Algorithm 37** and **Algorithm 38** are introduced to describe techniques to calculate $\mathrm{succ}\,(x)$ and $\mathrm{pred}\,(x)$, respectively:

Algorithm 37 succ

A calculation technique to find the largest floating-point number $y \in \mathbb{F}$ next to $x \in \mathbb{F}$.

function $y = \mathrm{succ}\,(x)$
  if $x \geqq 0$? $y = \mathrm{abssucc}(x)$: $y = -\mathrm{abspred}(-x)$;
end

Algorithm 38 pred

A calculation technique to find the smallest floating-point number $y \in \mathbb{F}$ next to $x \in \mathbb{F}$.

function $y = \mathrm{pred}\,(x)$
  if $x \geqq 0$? $y = \mathrm{abspred}(x)$: $y = -\mathrm{abssucc}(-x)$;
end

Note that abssucc and abspred are defined in **Algorithm 39** and **Algorithm 40** for non-positive inputs:

Algorithm 39 abssucc

A calculation technique to find the largest floating-point number $y \in \mathbb{F}$ in magnitude any positive floating-point number $x \in \mathbb{F}$.

function $y = \mathrm{abssucc}\,(x)$
  if $|x| \geqq 2^{-1022}$? $y = |x| \oslash \left(1 \ominus 2^{-53}\right)$: $y = |x| \oplus 2^{-1074}$;
end

Algorithm 40 abspred

A calculation technique to find the smallest floating-point number $y \in \mathbb{F}$ in magnitude next to any positive floating-point number $x \in \mathbb{F}$.

function $y = \mathrm{abspred}\,(x)$
  if $|x| \geqq 2^{-1022}$? $y = \left(1 \ominus 2^{-53}\right) \otimes |x|$: $y = |x| \oplus 2^{-1074}$;
end

Appendix A.4. Error-Free Transformation

An addition and multiplication of two floating-point numbers a and b can be converted without any error to the form of $x + y$, using the approximation x and its error. This type of conversion is called "error-free transformation". To describe TwoSum and TwoProduct below, we use MATLAB-like programming constructs. For example, the bracket $[\cdot, \cdot]$ means that an algorithm outputs two variables.

Error-free transformation on addition

Knuth proved that the addition of any two floating-point numbers a and b could be calculated without any errors ([6]).

Algorithm 41 Error-free transformation on addition

$$
\begin{aligned}
&\text{function } [a, b] = \text{TwoSum}(x, y)\\
&\quad a = x \oplus y\\
&\quad c = a \ominus x\\
&\quad b = (x \ominus (a \ominus c)) \oplus (y \ominus c)\\
&\text{end}
\end{aligned}
$$

Algorithm 41 indicates the following fact:

- x is the best approximation of $a + b$.
- The error of x, *i.e.* $y = a + b - x$, is a floating-pointing number.

Error-free transformation on multiplication

Dekker proved that error-free transformation could be conducted on multiplication ([7]).

Algorithm 42 Error-free transformation on multiplication

$$
\begin{aligned}
&\text{function } [a, b] = \text{TwoProduct}(x,\ y)\\
&\quad a = x \otimes y\\
&\quad [x_1, x_2] = \text{Split}(x)\\
&\quad [y_1, y_2] = \text{Split}(y)\\
&\quad b = x_2 \otimes y_2 \ominus (((a \ominus x_1 \otimes y_1) \ominus x_2 \otimes y_1) \ominus x_1 \otimes y_2)\\
&\text{end}
\end{aligned}
$$

Algorithm 43 describes Split required for TwoProduct.

Algorithm 43 Free-error splitting transformation of a floating-point number x.

function $[x_h, x_t] = \mathrm{Split}(x)$
$c = (2^{27} + 1) \otimes x$
$x_h = c \ominus (c \ominus x)$
$x_t = x \ominus x_h$
end